The Patent Office
PATENTS • DESIGNS • COPYRIGHT • TRADE MARKS

INVESTOR IN PEOPLE

PRIORITY DOCUMENT

SUBMITTED OR TRANSMITTED IN COMPLIANCE WITH RULE 17.1(a) OR (b)

The Patent Office
Concept House
Cardiff Road
Newport
South Wales
NP10 8QQ

REC'D **0 5 NOV 2004**

WIPO | PCT

I, the undersigned, being an officer duly authorised in accordance with Section 74(1) and (4) of the Deregulation & Contracting Out Act 1994, to sign and issue certificates on behalf of the Comptroller-General, hereby certify that annexed hereto is a true copy of the documents as originally filed in connection with the patent application identified therein.

In accordance with the Patents (Companies Re-registration) Rules 1982, if a company named in this certificate and any accompanying documents has re-registered under the Companies Act 1980 with the same name as that with which it was registered immediately before re-registration save for the substitution as, or inclusion as, the last part of the name of the words "public limited company" or their equivalents in Welsh, references to the name of the company in this certificate and any accompanying documents shall be treated as references to the name with which it is so re-registered.

In accordance with the rules, the words "public limited company" may be replaced by p.l.c., plc, P.L.C. or PLC.

Re-registration under the Companies Act does not constitute a new legal entity but merely subjects the company to certain additional company law rules.
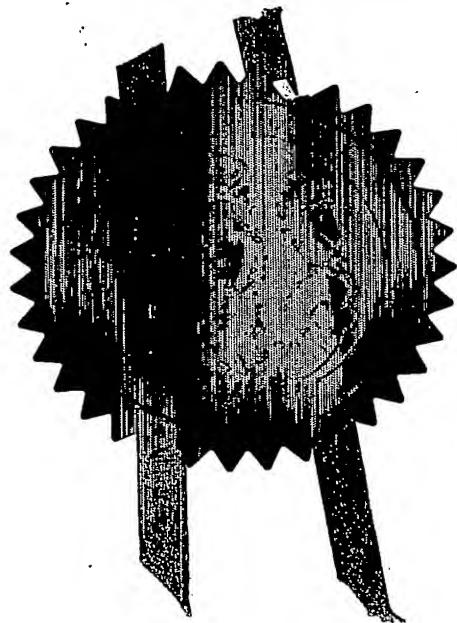
Signed *Evans*

Dated   21 October 2004

The **Patent Office**	Request for grant of a Patent
	Form 1/77 Patents Act 1977

1   Title of invention

Mark-up language framework with generic handling of data sources

2.   Applicant's details

0323439.0

[X]   First or only applicant

2a   If applying as a corporate body: Corporate Name

Symbian Limited

Country

GB

2b   If applying as an individual or partnership
Surname

Forenames

2c   Address   Sentinel House
16 Harcourt Street
London

UK Postcode   W1H 1DS

Country   GB

ADP Number   07573132002

☐ **2d**	**Second applicant (if any)** **Corporate Name** Country
2e	Surname Forenames
2f	Address UK Postcode Country ADP Number
3 Address for service	Agent's Name Origin Limited Agent's Address 52 Muswell Hill Road London Agent's postcode N10 3JR Agent's ADP C03274 Number

4 Reference Number

Generic Data (UK)

5 Claiming an earlier application date

An earlier filing date is claimed:

Yes ☐   No ☒

Number of earlier
application or patent number

Filing date

15 (4) (Divisional)   8(3)   12(6)   37(4)

☐   ☐   ☐   ☐

6 Declaration of priority

Country of filing	Priority Application Number	Filing Date

7 Inventorship

The applicant(s) are the sole inventors/joint inventors

Yes ☐   No ☒

8 Checklist

Continuation sheets

Claims   1 ✓   Description   38 ✓

Abstract   1 ✓   Drawings   6 + 6 *sheets*

Priority Documents   Yes/No

Translations of Priority Documents   Yes/No

Patents Form 7/77   Yes/No

Patents Form 9/77   Yes/No ✓

Patents Form 10/77   Yes/No

9 Request

We request the grant of a patent on the basis
of this application

Signed: *Origin Limited*   Date: 7 October 2003

(Origin Limited)

MARK-UP LANGUAGE FRAMEWORK WITH GENERIC HANDLING OF DATA SOURCES

DESCRIPTION OF THE PRIOR ART

5   Mark-up language is a set of codes in a text file that enable a computing device to format the text for correct display or printing. A client (i.e. any process that requests a service from another process) in a software system creates mark-up language using a 'generator'. It reads and interprets mark-up language using a 'parser'.

In the prior art, parsers and generators have been specific to certain kinds of mark-up

10   languages. For example, a client could use an XML (extensible mark-up language) parser to interpret and handle XML files; it could use a separate WBXML (WAP binary XML) parser to interpret and handle WBML files. In each case, the client would talk *directly* to each parser.

When the client needs to generate mark-up language format files, there could be an XML

15   generator and a separate WBXML generator. Again, the client would talk *directly* to each generator.

In prior art systems, the kind of data source that was used by the parser or generator was fixed; parsers and generators were hard-coded to use a specific type of source – e.g. a buffer in memory.

20

SUMMARY OF THE PRESENT INVENTION

The present invention is a portable computing device programmed with a mark-up language parser or generator that can access data from a source using a generic data supplier API.

25

Hence, the parser or generator is insulated from having to talk directly to a data source; instead, it does so via a generic data supplier API, acting as an intermediary layer. This

de-couples the parser or generator from the data source and hence means that the parser or generator no longer have to be hard coded for a specific data supplier. This in turn leads to a simplification of the parser and generator design.

The present invention allows parsing and generation to be carried out with any data source. For example, a buffer in memory could be used, as could a file, as could streaming from a socket (hence enabling the ability to parse in real-time from data streamed over the internet). There is no requirement to define, at parser/generator build time, what particular data source will be used. Instead, the system allows any source that can use the generic data supplier API to be adopted. New types of data sources can be utilised by computing device, even after those devices have been shipped to end-users.

DETAILED DESCRIPTION

Overview of Key Features

The present invention is implemented in a system called the Mark-Up Language Framework, used in SymbianOS from Symbian Limited, London, United Kingdom.

5   SymbianOS is an operating system for smart phones and advanced mobile telephones, and other kinds of portable computing devices.

The Mark-Up Language framework implements three key features.

1.   Generic Parser API

Clients are separated from mark-up language parsers/generators by an intermediary layer

10   that (a) insulates the client from having to communicate directly with the parser or generator and is (b) generic in that it presents a common API to the client irrespective of the specific kind of parser or generator the intermediary layer interfaces with.

2.   Data validation/pre-filtering and altering components in a chain of responsibility

15   Mark-up language parsers or generators can access components to validate, pre-filter or alter data; the components are plug-in components that operate using a 'chain of responsibility' design pattern.

3.   Generic Data Supplier API

The mark-up language parsers or generators can access data from a source using a

20   generic data supplier API, insulating the parser or generator from having to communicate directly with the data source.

Each of this features will now be discussed in more detail.

1.   Generic Parser Intermediary Layer

The essence of this approach is that the client that interfaces with a mark up language

25   parser or a generator via an intermediary layer that (a) insulates the client from having to communicate directly with the parser or generator and is (b) generic in that it presents a

common API to the client irrespective of the specific kind of parser or generator the intermediary layer interfaces with.

In this way, the client is no longer tied to a single kind of parser or generator; it can operate with any different kind of parser compatible with the intermediary layer, yet it

5   remains far simpler than prior art clients that are hard-coded to operate directly with several different kinds of parsers and generators.

The API is typically implemented as a header file. In an implementation, the intermediary layer acts as an extensible framework and the parsers and generators are themselves plug-ins to that framework. The present invention may hence readily allow

10   the device to operate with different kinds of parsers and generators: this extensibility is impossible to achieve with prior art hard-coded systems.

The specific kind of parser or generator being used is not known to the client: the intermediary layer fully insulates the client from needing to be aware of these specifics. Instead, the client deals only with the intermediary layer, which presents to the client as a

15   generic parser or a generic generator – i.e. a parser or generator which behaves in a way that is common to all parsers or generators.

For example, the SyncML the protocol supports both XML and WBXML. By using both XML and WBXML parser and generator plug-ins in to the framework, a SyncML client can use either or both type of parser and generator without knowing about the type of

20   mark-up language; as a result, the design of the SyncML client is greatly simplified. Since WBXML and XML are quite different in the way they represent their data, one very useful feature of the invention is the mapping of WBXML tokens to a string in a static string pool table. Appendix B expands on this idea.

The present invention may provide a flexible and extensible file conversion system: for

25   example, the device could parse a document written in one mark up language format and then use the parsed document data to generate an equivalent document in a different file format. Because of the extensible plug-in design of an implementation of the system, it is possible to provide far greater kinds of file conversion capabilities than was previously the case. New kinds of parsers and generators can be provided for loading onto a device

30   after that device has been shipped to an end-user. The only requirement is that they are compatible with the intermediary layer.

Another advantage of the present invention is that it allows not only different parsers and generators to be readily used by the same client, but it allows also several different clients to share the same parsers and generators as well. The API may itself be

5   extensible, so that extensions to its capabilities (e.g. to enable a new/extended mark-up language of a document to be handled) can be made without affecting compatibility with existing clients or existing parsers and generators. Similarly, new kinds of clients can be provided for loading onto a device after that device has been shipped to an end-user. The only requirement is that they are compatible with the intermediary layer.

10

2.   **Data validation/pre-filtering and altering components in a chain of responsibility**

The essence of this approach is that the mark-up language parser or generator can access components to validate, pre-filter or alter data, in which the components are plug-in

15   components that operate using a chain of responsibility.

Because of the plug-in design of the components, the system is inherently flexible and extensible compared with prior art systems in which a component (for validating, pre-filtering or altering data from a parser or generator) would be tied exclusively to a given parser. Hence, if a mark up language of a document is extended, or a new one created, it

20   is possible to write any new validation/pre-filter/altering plug-in that is needed to work with the extended or new language. These new kinds of validation/pre-filter/altering plug-ins can be provided for loading onto a device even after that device has been shipped to an end-user. The 'chain of responsibility' design pattern, whilst known in object oriented programming, has not previsouly been used in the present context.

25   The plug-in components may all present a common, generic API to the parser and generator. Hence, the same plug-in can be used with different types of parsers and generators (e.g. a XML parser, a WBXML parser, a RTF parser etc.). The plug-ins also present a common, generic API to a client component using the parser or generator. Hence, the same plug-ins can be used by different clients.

30   For example a DTD validator plug-in could be written that validates the mark-up of a document and can report errors to the client. Or for a web browser an auto correction

plug-in filter could be written that tries to correct errors found in the mark-up language, such as a missing end element tag, or a incorrectly placed element tag. The auto correction plug-in will, if it can, fix the error transparently to the client. This enables a web browser to still display a document rather then just displaying an error reporting that

5   there was an error in the document.

Because, the plug-ins can be chained together, complex and different type of filtering and validation can take place. In the example above the parser could notify the validator plug-in of elements it is parsing and these in turn would go to the auto correction plug-in to

10.   be fixed if required and finally the client would receive these events.

The mark-up framework allows parser plug-ins to expose the parsed element stack to all validation/pre-filter/altering plug-ins. (The parsed element stack is a stack populated with elements from a document extracted as that document is parsed; this stack is made

15   available to all validation/pre-filter/altering plug-ins to avoid the need to duplicate the stack for each of these plug-ins).   This also enables the plug-ins to use the stack information to aid in validation and filtering. For example an auto corrector plug-in may need to know the entire element list that is on the stack in order to figure out how to fix a problem.

20

The use of filter/validator plug-ins in mark-up language generators is especially useful for developers writing a client to the framework and generating mark-up documents as the same validator plug-in used by the parser can be used in the generator. Errors are reported to the client when the mark-up does not conform to the validator which will

25   enable the developer to make sure they are writing well formed mark-up that conforms to the DTD and catch error early on during development.

The mark-up framework incorporates a character conversion module that enables documents written in different character sets (e.g, ASCII, various Kanji character sets

30   etc.) to be parsed and converted to UTF8. This means a client obtains the results from the parser in a generic way (UTF8) without having to know the original character set that was used in the document. Clients hence no longer need to be able to differentiate between different character sets and handle the different character sets appropriately.

3. Generic Data Supplier API

The mark-up language parser or generator accesses data from a source using a generic data supplier API. Hence, the parser or generator is insulated from having to talk directly to a data source; instead, it does so via a generic data supplier API, acting as an

5   intermediary layer. This de-couples the parser or generator from the data source and hence means that the parser or generator no longer have to be hard coded for a specific data supplier. This in turn leads to a simplification of the parser and generator design.

The present invention allows parsing and generation to be carried out with any data source. For example, a buffer in memory could be used, as could a file, as could

10   streaming from a socket (hence enabling the ability to parse in real-time from data streamed over the internet). There is no requirement to define, at parser/generator build time, what particular data source will be used. Instead, the system allows any source that can use the generic data supplier API to be adopted. New types of data sources can be utilised by computing device, even after those devices have been shipped to end-users.

15

The present invention is implemented in a system called the Mark-Up Language Framework, used in SymbianOS from Symbian Limited, London, United Kingdom. SymbianOS is an operating system for smart phones and advanced mobile telephones, and other kinds of portable computing devices.

20

Appendix 1 describes the Mark-Up Language Framework in more detail. **Appendix 2** describes a particular technique, referred to as 'String Pool', which is used in the Mark-Up Language Framework. The appendices refer to various SymbianOS specific programming techniques and structures. There is an extensive published literature

25   describing these techniques; reference may for example be made to "Professional Symbian Programming" Wrox Press Inc. ISBN: 186100303X.

8

Appendix 1

Mark-Up Framework Design Document v0.4

5

Introduction

Purpose and Scope

This document describes the architecture for a generic mark-up framework. The framework is extendable by using plug-ins so that mark-up parsers and generators (e.g.

5   XML[1], WBXML[2]) can be used.

Design Overview

Block Diagrams

The mark-up framework block diagram is shown in **Error! Reference source not**

10   **found..** The **Client** is the application using the mark-up framework for parsing or generating a document. The **Parser** and **Generator** components are ECOM plug-ins specific to a mark-up language (e.g. XML or WBXML). These components use the **Namespace collection** to retrieve information about a specific namespace during the parsing or generating phase.

15   The **Namespace Plug-in** component is an ECOM plug-in that sets-up all the elements, attributes and attribute values for a namespace. For each namespace used there must be a plug-in that describes the namespace. The namespace information is stored in a string pool. The string pool is a way of storing strings that makes comparison almost instantaneous at the expense of string creation. It is particularly efficient at handling

20   string constants that are known at compile time, which makes it very suitable for processing documents. The **Namespace** owns the string pool that the **Parser, Generator** and **Client** can gain access to.

The **Namespace Plug-in** simply sets-up the string pool with the required strings for the namespace the plug-in represents. The **Client** may get access to the **Namespace**

25   **Collection** via the **Parser** or **Generator** to pre-load namespaces prior to parsing or generating documents which may speed up the parsing or generating session.

The Plug-in components (1 – 4) are optional and allow further processing of the data before the client receive it such as DTD validators or document auto correctors.

30   Validators check the elements and attributes conform to the DTD. Document auto correction plug-ins are used to try to correct errors reported from DTD validators.

The parser is event driven and sends events to the various plug-ins and UI during parsing. **Error! Reference source not found.** shows a client parsing with a DTD validator and auto corrector. The client talks to the parser directly to start the parse. The

parser sends events to the chain of plug-ins. The first plug-in that receives events is the DTD validator plug-in. This plug-in validates that the data in the event it received is correct. If it is not correct it will send the same event the parser sent to the validator to the auto corrector except for a error code that will describe the problem the validator

5   encountered. It the event data is valid the same event will be sent to the auto corrector. Now the auto corrector receives the event and can check for any errors. If there is an error it can attempt to correct it. If it can correct the error it will modify the data in the event and remove the error code before sending the event to the client. The client finally receives the event and can now handle it.

10   **Error! Reference source not found.** illustrates a client generating using a DTD validator and auto corrector plug-ins. A real client would probably never use a generator and auto corrector since the data the client generates should always be valid, but it is used here to show the flow of events from a generator and any plug-ins attached.

15   The client sends a build request to the generator. The first thing the generator does is to send the request as an event to the DTD validator plug-in. The situation is similar to the parser, the DTD validator plug-in validates that the data in the event it received is correct. If it is not correct it will send the same event the parser sent to the validator to the auto corrector except for a error code that will describe the problem the validator

20   encountered. It the event data is valid the same event will be sent to the auto corrector. Now the auto corrector receives the event and can check for any errors. If there is an error it can attempt to correct it. If it can correct the error it will modify the data in the event and remove the error code before sending the event back to the generator. The major difference between the events during parsing and generating is with generating,

25   once the final plug-in has dealt with the event it gets sent back to the generator. The generator receives the event and builds up part of the document using the details from the event.

30   **Parsing and Generating WBXML**
Parsing WBXML is quite different to parsing XML or HTML. The main difference is elements and attributes are defined as tokens rather than using their text representation. This means a mapping needs to be stored between a WBXML token and its static string

representation. The **Namespace plug-in** for a particular namespace will store these mappings. A WBXML parser and generator can then obtain a string from the namespace plug-in given the WBXML token and vice versa.

5   Class Diagram

The class diagram for the mark-up framework is shown in **Error! Reference source not found.**. The diagram also depicts plug-ins that makes use of the framework. The green (or dark grey classes in b&w) are the plug-ins that provide implementation to the mark-up framework. **CXmlParser** and **CWbxmlParser** provide an implementation to parse
10 XML and WBXML documents respectively. In the same way **CXmlGenerator** and **CWbxmlGenerator** generate XML and WBXML documents respectively. **CValidator** is a plug-in which will validate the mark-up document during parsing or generating. **CAutoCorrector** is a plug-in that corrects invalid mark-up documents.

When parsing a document and the client receives events for the start of an element for
15 example (**OnStartElementL**), the element **RString** in the event is a handle to a string in the string pool. If this is a known string, i.e. one that has been added by the **Namespace Plug-in** then the string will be static. Otherwise, if it is an unknown string, the parser will add the string to the string pool as a dynamic string and return a **RString** with a handle of this string. It is not possible to know if a **RString** is dynamic or static so the parser or
20 generator that obtains a **RString** must be sure to close it to ensure any memory is released if the string is dynamic. A client that wishes to use the **RString** after the event returns to the parser must make a copy of it which will increase the reference count and make sure it is not deleted when the parser closes it.

25 **Error! Reference source not found.** is an example class diagram that shows the major classes for parsing WBXML SyncML documents. The client creates a **CDescriptorDataSupplier** that supplies the data to the parser. **CWbxmlParser** is the class that actually parses the document. **CSyncMLNamespace** is the namespace for SyncML that the parser uses to map WBXML tokens to strings. All the other classes
30 belong to the mark-up framework. To parse a document with different namespaces the only thing that needs to be added is a plug-in for each namespace.

Class Dictionary

Object name	Description	Associated (owned/dependant) objects
MMarkupCallback	A call-back that a client must implement so that the parser can report events back to the client during the parsing session.	Inherited by clients and plug-ins.
RNamespaceCollection	Contains a collection of namespaces. Contains reference counter so multiple parsers or generators may use the same namespace collection.	Owned by either **CParserSession** or **CGeneratorSession**. Owns an array of **CMarkupNamespace** plug-ins.
CMarkupNamespace	ECOM interface to implement a namespace.	Inherited by any namespace plug-ins.
RParserSession	Public interface for a client to create a parser session.	Owned by the client.
RGeneratorSession	Public interface for a client to create a generator session.	Owned by the client.
CMarkupCharSetConverter	Helper function which uses **CCnvCharacterSetConverter** for the client, parser and generator to do any character set conversions or resolving MIB Enums or Internet-standard names of character sets.	Owned by **RParserSession** and RGeneratorSession.
CMarkupPluginBase	Generic interface for any type of plug-in.	Inherited by **CMarkupPlugin**, **CParserSession** and **CGeneratorSession**.
CMarkupPlugin	ECOM interface for plug-ins to be used by the parser and generator.	Owned by **CParserSession** or **CGeneratorSession**.
MDataSupplierReader	Pure virtual interface to be implemented by a data supplier for reading data.	Inherited by the client's data provider.
MDataSupplierWriter	Pure virtual interface to be implemented by a data supplier for writing data.	Inherited by the client's data provider.
CParserSession	ECOM interface for parser plug-ins.	Inherited by a concrete parser implementation.
CGeneratorSession	ECOM interface for generator plug-ins.	Inherited by a concrete generator implementation.
RAttribute	Contains the name and value of an attribute.	Used by the parse, generator and client.

The classes below are not part of the framework but illustrate how the framework can be used.		
CValidator	A DTD, schema or some other type of validator.	Owned by **RParserSession** or **RGeneratorSession.**
CAutoCorrector	Used to auto correct invalid data.	Owned by **RParserSession** or **RGeneratorSession.**
CXmlParser	An XML parser implementation.	Owned by **RParserSession.**
CWbxmlParser	A WBXML parser implementation.	Owned by **RParserSession.**
CXmlGenerator	An XML generator implementation.	Owned by RGeneratorSession.
CWbxmlGenerator	A WBXML generator implementation.	Owned by **RGeneratorSession.**
CNamespace	A namespace plug-in to use with a parser and generator.	O w n e d b y **RNamespaceCollection.**
RElementStack	A stack of the currently processed elements during parsing or generating.	Owned by **CParserSession** and **CGeneratorSession.**

Detailed Design

5   **RParserSession**

The following is the public API for this class:

Method	Description
void OpenL(MDataSupplierReader& **aReader,** c o n s t T D e s C 8 & **aMarkupMimeType,** c o n s t T D e s C 8 & **aDocumentMimeType,** MMarkupCallback& **aCallback)**	Opens a parser session. **aReader** is the data supplier reader to use during parsing. **aMarkupMimeType** is the MIME type of the parser to open. **aDocumentMimeType** is the MIME type of the document to parse. **aCallback** is a reference to the call-back so the parser can report events.
void OpenL(	Opens a parser session.

MDataSupplierReader& **aReader,** const T D e s C 8 & **aMarkupMimeType,** const T D e s C 8 & **aDocumentMimeType,** MMarkupCallback& **aCallback,** RMarkupPlugins **aPlugins)**	**aReader** is the data supplier reader to use during parsing. **aMarkupMimeType** is the MIME type of the parser to open. **aDocumentMimeType** is the MIME type of the document to parse. **aCallback** is a reference to the call-back so the parser can report events. **aPlugins** is an array of plug-ins to use with the parser. The first plug-in in the list is the first plug-in to be called back from the parser. The first plug-in will then call-back to the second plug-in etc.
void OpenL(MDataSupplierReader& **aReader,** const T D e s C 8 & **aMarkupMimeType,** const T D e s C 8 & **aDocumentMimeType,** MMarkupCallback& **aCallback,** RMarkupPlugins **aPlugins[],** RNamespaceCollection **aNamespaceCollection)**	Opens a parser session. **aReader** is the data supplier reader to use during parsing. **aMarkupMimeType** is the MIME type of the parser to open. **aDocumentMimeType** is the MIME type of the document to parse. **aCallback** is a reference to the call-back so the parser can report events. **aPlugins** is an array of plug-ins to use with the parser. The first plug-in in the list is the first plug-in to be called back from the parser. The first plug-in will then call-back to the second plug-in etc. **aNamespaceCollection** is a handle to a previous namespace collection. This is useful if a generator or another parser session has been created so that same namespace collection can be shared.
void Close()	Closes the parser session.
void Start()	Start parsing the document.
void Stop()	Stop parsing the document.
void Reset(MDataSupplierReader& **aReader,** MMarkupCallback& **aCallback)**	Resets the parser ready to parse a new document. **aReader** is the data supplier reader to use during parsing. **aCallback** is a reference to the call-back so the parser can

	report events.
TInt SetParseMode(TInt aParseMode)	Selects one or more parse modes. **aParseMode** is one or more of the following: 　　**EConvertTagsToLowerCase** – Converts elements and attributes to lowercase. This can be used for case-insensitive HTML so that a tag can be matched to a static string in the string pool. 　　**EErrorOnUnrecognisedTags** - Reports an error when unrecognised tags are found. 　　**EReportUnrecognisedTags** - Reports unrecognised tags. 　　**EReportNamespaces** – Reports the namespace. 　　**EReportNamespacePrefixes** – Reports the namespace prefix. 　　**ESendFullContentInOneChunk** – Sends all content data for an element in one chunk. 　　**EReportNameSpaceMapping** – Reports namespace mappings via the **DoStartPrefixMapping()** & **DoEndPrefixMapping()** methods. If this function is not called the default will be: **EReportUnrecognisedTags | EReportNamespaces** If the parsing mode is not supported **KErrNotSupported** is returned.

RGeneratorSession

The following is the public API for this class:

Method	Description
void OpenL(MDataSupplierWriter& aWriter,	Opens a generator session. **aWriter** is the data supplier writer used to generate a

TUid **aMarkupMimeType,** c o n s t T D e s C 8 & **aDocumentMimeType)**	document. **aMarkupMimeType** is the MIME type of the generator to open. **aDocumentMimeType** is the MIME type of the document to parse.
void OpenL(MDataSupplierWriter& **aWriter,** TUid **aMarkupMimeType,** c o n s t T D e s C 8 & **aDocumentMimeType,** RMarkupPlugins **aPlugins**[])	Opens a generator session. **aWriter** is the data supplier writer used to generate a document. **aMarkupMimeType** is the MIME type of the generator to open. **aDocumentMimeType** is the MIME type of the document to parse. **aPlugins** is an array of plug-ins to use with the generator.
void OpenL(MDataSupplierWriter& **aWriter,** TUid **aMarkupMimeType,** c o n s t T D e s C 8 & **aDocumentMimeType,** RMarkupPlugins **aPlugins**[], RNamespaceCollection **aNamespaceCollection)**	Opens a generator session. **aWriter** is the data supplier writer used to generate a document. **aMarkupMimeType** is the MIME type of the generator to open. **aDocumentMimeType** is the MIME type of the document to parse. **aPlugins** is an array of plug-ins to use with the generator. **aNamespaceCollection** is a handle to a previous namespace collection. This is useful if a generator or another parser session has been created so that same namespace collection can be shared.
void Close()	Closes the generator session.
void Reset(MDataSupplierWriter& **aWriter,** MMarkupCallback& **aCallback)**	Resets the generator ready to generate a new document. **aWriter** is the data supplier writer used to generate a document. **aCallback** is a reference to the call-back so the generator can report events.
void BuildStartDocumentL(RDocumentParameters	Builds the start of the document. **aDocParam** specifies the various parameters of the

aDocParam);	document. In the case of WBXML this would state the public ID and string table.
void BuildEndDocumentL()	Builds the end of the document.
void BuildStartElementL(RTagInfo& aElement, RAttributeArray& aAttributes)	Builds the start element with attributes and namespace if specified. aElement is a handle to the element's details. aAttributes contains the attributes for the element.
void BuildEndElementL(RTagInfo& aElement)	Builds the end of the element. aElement is a handle to the element's details.
void BuildContentL(const TDesC8& aContentPart)	Builds part or all of the content. Large content should be built in chunks. I.e. this function should be called many times for each chunk. aBytes is the raw content data. This data must be converted to the correct character set by the client.
void BuildPrefixMappingL(RString& aPrefix, RString& aUri)	Builds a prefix – URI namespace for the next element to be built. This method can be called for each namespace that needs to be declared. aPrefix is the Namespace prefix being declared. aUri is the Namespace URI the prefix is mapped to.
void BuildProcessingInstructionL(RString& aTarget, RString& aData)	Build a processing instruction. aTarget is the processing instruction target. aData is the processing instruction data.

RTagInfo

The following is the public API for this class:

Method	Description
void Open(RString& aUri, RString& aPrefix, RString& aLocalName)	Sets the tag information for an element or attribute. aUri is the URI of the namespace. aPrefix is the prefix of the qualified name. aLocalName is the local name of the qualified name.
void Close()	Closes the tag information.
RString& Uri()	Returns the URI.
RString& LocalName()	Returns the local name.

RString& Prefix()	Returns the prefix.

RNamespaceCollection

The following is the public API for this class:

Method	Description
void Connect()	Every time this method is called a reference counter is incremented so that the namespace collection is only destroyed when no clients are using it.
void Close()	Every time this method is called a reference counter is decremented and the object is destroyed only when the reference counter is zero.
const CMarkupNameSpace& OpenNamespaceL(const TDesC8& aMimeType)	Opens a namespace plug-in and returns a reference to the namespace plug-in. If the namespace plug-in is not loaded it will be automatically loaded. aMimeType is the MIME type of the plug-in to open.
const CMarkupNameSpace& OpenNamespaceL(TUint8 aCodePage)	Opens a namespace plug-in and returns a reference to the namespace plug-in. aCodePage is the code page of the plug-in to open.
void Reset()	Resets the namespace collection and string pool.
RStringPool StringPool()	Returns a handle to the string pool object.

5   CMarkupNamespace

The following is the API for this class:

Method	Description
void NewL(RStringPool aStringPool)	Creates the namespace plug-in. aStringPool is a handle of the string pool to add static string tables.
RString& Element(TUint8 aWbxmlToken) const	Returns a handle to the string. aWbxmlToken is the WBXML token of the element.
void AttributeValuePair(TUint8 aWbxmlToken RString& aAttribute, RString& aValue) const	Returns a handle to the attribute and value strings. aWbxmlToken is the WBXML token of the attribute. aAttribute is the handle to the attribute string. aValue is the handle to the value string.

RString& AttributeValue(TUint8 **aWbxmlToken**) const	Returns a handle to an attribute value. **aWbxmlToken** is the WBXML token of the attribute.
RString& NamespaceUri() const	Returns the namespace name.
TUint8 CodePage() const	Returns the code page for this namespace.

RTableCodePage

The following is the API for this class:

Method	Description
RString NameSpaceUri()	Returns the namespace URI for this code page.
TInt StringPoolIndexFromToken(TInt **aToken**);	Gets a StringPool index from a token value. –1 is returned if the item is not found.
TInt TokenFromStringPoolIndex(TInt **aIndex**);	Gets a token value from a StringPool index. –1 is returned if the item is not found.

5   CMarkupPluginBase

The following is the API for this ECOM class:

Method	Description
CMarkupPluginBase& RootPlugin()	Returns a reference to the root plug-in. This must be either a parser or generator plug-in.
CMarkupPluginBase& ParentPlugin()	Returns a reference to the Parent plug-in.
RElementStack& ElementStack()	Returns a handle to the element stack.
RNameSpaceCollection& NamespaceCollection()	Returns a handle to the namespace collection.
CMarkupCharSetConverter& CharSetConverter()	Returns a reference to the character set converter object.
TBool IsChildElementValid(RString& **aParentElement**, RString& **aChildElement**)	Checks if the **aChildElement** is a valid child of **aParentElement**.

CMarkupPlugin

The following is the API for this ECOM class:

Method	Description

CMarkupPlugin* NewL(MMarkupCallback& aCallback)	Creates an instance of a mark-up plug-in. aCallback is a reference to the call-back to report events.
void SetParent(CMarkupPluginBase* aParentPlugin)	Sets the parent plug-in for this plug-in. aParentPlugin is a pointer to the parent plug-in or NULL if there is no parent. A parser or generator does not have a parent so this must not be set, as the default NULL will indication there is not parent.

CParserSession

The following is the API for this ECOM class:

Method	Description
CParserSession* NewL(MDataSupplierReader& aReader, const TDesC8& aMarkupMimeType, const TDesC8& aDocumentMimeType, MMarkupCallback& aCallback, RNamespaceCollection* aNamespaceCollection, CMarkupCharSetConverter& aCharSetConverter)	Opens a parser session. aReader is the data supplier reader to use during parsing. aMarkupMimeType is the MIME type of the parser to open. aDocumentMimeType is the MIME type of the document to parse. aCallback is a reference to the call-back so the parser can report events. aNamespaceCollection is a handle to a previous namespace collection. Set to NULL if a new RNamespaceCollection is to be used. aCharSetConverter is a reference to the character set conversion class.
void Start()	Start parsing the document.
void Stop()	Stop parsing the document.
void Reset(MDataSupplierReader& aReader, MMarkupCallback& aCallback)	Resets the parser ready to parse a new document. aReader is the data supplier reader to use during parsing. aCallback is a reference to the call-back so the parser can report events.
void SetParseMode(TInt aParseMode)	Selects one or more parse modes. See RParserSession for details on aParseMode.

CGeneratorSession

The following is the API for this ECOM class:

Method	Description
void OpenL(MDataSupplierWriter& aWriter, TUid **aMarkupMimeType,** c o n s t T D e s C 8 & **aDocumentMimeType,** MMarkupCallback& aCallback, RNamespaceCollection* **aNamespaceCollection,** CMarkupCharSetConverter& **aCharSetConverter)**	Opens a generator session. **aWriter** is the data supplier writer used to generate a document. **aMarkupMimeType** is the MIME type of the generator to open. **aDocumentMimeType** is the MIME type of the document to parse. **aCallback** is a reference to the call-back so the generator can report events. **aNamespaceCollection** is a handle to a previous namespace collection. Set to NULL if a new **RNamespaceCollection** is to be used. **aCharSetConverter** is a reference to the character set conversion class.
void Reset(MDataSupplierWriter& **aWriter,** MMarkupCallback& **aCallback)**	Resets the generator ready to generate a new document. **aWriter** is the data supplier writer used to generate a document. **aCallback** is a reference to the call-back so the generator can report events.
void BuildStartDocumentL(RDocumentParameters **aDocParam)**;	Builds the start of the document. **aDocParam** specifies the various parameters of the document.
void BuildEndDocumentL()	Builds the end of the document.
void BuildStartElementL(RTagInfo& **aElement,** RAttributeArray& **aAttributes)**	Builds the start element with attributes and namespace if specified. **aElement** is a handle to the element's details. **aAttributes** contains the attributes for the element.
void BuildEndElementL(RTagInfo& **aElement)**	Builds the end of the element. **aElement** is a handle to the element's details.
void BuildContentL(	Builds part or all of the content. Large content should be

const TDesC8& aContentPart)	built in chunks. I.e. this function should be called many times for each chunk. aBytes is the raw content data. This data must be converted to the correct character set by the client.
void BuildProcessingInstructionL(RString& aTarget, RString& aData)	Build a processing instruction. aTarget is the processing instruction target. aData is the processing instruction data.

RAttribute

The following is the API for this class:

Method	Description
RTagInfo& Attribute()	Returns a handle to the attribute's name details.
TAttributeType Type()	Returns the attribute's type. Where TAttributeType is one of the following enumeration: CDATA ID IDREF IDREFS NMTOKEN NMTOKENS ENTITY ENTITIES NOTATION
RString& Value()	Returns a handle to the attribute value. If the attribute value is a list of tokens (IDREFS, ENTITIES or NMTOKENS), the tokens will be concatenated into a single RString with each token separated by a single space.

5

MDataSupplierReader

The following is the API for this mix-in class:

Method	Description

TUint8 GetByteL()	Get a single byte from the data supplier.
const TDesC8& GetBytesL(TInt aNumberOfBytes)	Gets a descriptor of size aNumberOfChars. If the number of bytes is not available this method leaves with KErrEof. The returned descriptor must not be deleted until another call to GetBytesL or EndTransactionL() is made.
void StartTransactionL()	The parser calls this to indicate the start of a transaction.
void EndTrasactionL()	The parser calls this to indicate the transaction has ended. Any data stored for the transaction may now be deleted.
void RollbackL()	The parse calls this to indicate the transaction must be rolled back to the exact state as when StartTransactionL() was called.

MDataSupplierWriter

The following is the API for this mix-in class:

Method	Description
void PutByteL(TUint8 aByte)	Put a byte in the data supplier.
void PutBytesL(const TDesC8& aBytes)	Puts a descriptor in the data supplier.

5   **MMarkupCallback**

The following is the API for this mix-in class:

Method	Description
void OnStartDocumentL(RDocumentParameters aDocParam, TInt aErrorCode);	Callback to indicate the start of the document. **aDocParam** specifies the various parameters of the document. **aErrorCode** is the error code. If this is not **KErrNone** then special action may be required.
void OnEndDocumentL(TInt aErrorCode);	Indicates the end of the document has been reached **aErrorCode** is the error code. If this is not **KErrNone** then special action may be required.
void OnStartElementL(RTagInfo& aElement,	Callback to indicate an element has been parsed. **aElement** is a handle to the element's details.

RAttributeArray& aAttributes, TInt aErrorCode);	aAttributes contains the attributes for the element. aErrorCode is the error code. If this is not **KErrNone** then special action may be required.
void OnEndElementL(RTagInfo& aElement, TInt aErrorCode);	Callback to indicate the end of the element has been reached. aElement is a handle to the element's details. aErrorCode is the error code. If this is not **KErrNone** then special action may be required.
void OnContentL(const TDesC8& aBytes, TInt aErrorCode)	Sends the content of the element. Not all the content may be returned in one go. The data may be sent in chunks. When an **OnEndElementL** is received this means there is no more content to be sent. **aBytes** is the raw content data for the element. The client is responsible for converting the data to the required character set if necessary. In some instances with WBXML opaque data the content may be binary and must not be converted. **aErrorCode** is the error code. If this is not **KErrNone** then special action may be required.
void OnStartPrefixMappingL(RString& aPrefix, RString& aUri, TInt aErrorCode)	Notification of the beginning of the scope of a prefix-URI Namespace mapping. This method is always called before the corresponding OnStartElementL method. **aPrefix** is the Namespace prefix being declared. **aUri** is the Namespace URI the prefix is mapped to. **aErrorCode** is the error code. If this is not **KErrNone** then special action may be required.
void OnEndPrefixMappingL(RString& aPrefix, TInt aErrorCode)	Notification of the end of the scope of a prefix-URI mapping. This method is called after the corresponding DoEndElementL method. **aPrefix** is the Namespace prefix that was mapped. **aErrorCode** is the error code. If this is not **KErrNone** then special action may be required.
void OnIgnoreableWhiteSpaceL(const TDesC8& aBytes,	Notification of ignorable whitespace in element content. **aBytes** are the ignored bytes from the document being

TInt aErrorCode)	parsed.
	aErrorCode is the error code. If this is not **KErrNone** then special action may be required.
void OnSkippedEntityL(RString& aName, TInt aErrorCode)	Notification of a skipped entity. If the parser encounters an external entity it does not need to expand it – it can return the entity as aName for the client to deal with. aName is the name of the skipped entity. aErrorCode is the error code. If this is not **KErrNone** then special action may be required.
void OnProcessingInstructionL(const TDesC8& aTarget, const TDesC8& aData, TInt aErrorCode)	Receive notification of a processing instruction. aTarget is the processing instruction target. aData is the processing instruction data. If empty none was supplied. aErrorCode is the error code. If this is not **KErrNone** then special action may be required.
void OnOutOfDataL()	There is no more data in the data supplier to parse. If there is more data to parse **Start()** should be called once there is more data in the supplier to continue parsing.
void OnError(TInt aError)	An error has occurred where **aError** is the error code

Sequence Diagrams

Setting up, parsing and generating

Error! Reference source not found. shows the interaction of the client with the various

5   parser objects to create a parser and generator session. The parsing of a simple document with only one element and generation of one element is shown. It is assumed a DTD validator and auto correct component are used. Auto correction in this example is only used with the parser. The generator only checks that tags are DTD compliant but does not try to correct any DTD errors.

10

Element not valid at current level in DTD

Auto correction is left up to the plug-in implementers to decide how and what should be corrected.

The sequence diagram in Figure 4 shows an example of what is possible with the case where the format of the document is valid, however, there is a invalid element (C) that should be at a different level as shown in an example document below:

```
5    <A>Content
          <B>
              <C> // Not valid for the DTD, should be a root element.
              Some content
              </C>
10        </B>
     </A>
     // <C> should go here
```

The bad element is detected by the DTD validator and sent to the auto correct
15 component. The auto corrector realises that this element has an error from the error code passed in the call-back and tries to find out where the element should go, and send back the appropriate **OnEndElementL()** call-backs to the client.

Scenarios
20 **Set-up a parser to parse WBXML without any plug-ins.**

Scenario to parse the following document:

```
     <A>
25        <B>
          Content
          </B>
     <A>
```

30.

   1. The client creates a data supplier that contains the data to be parsed.

2. The client creates an RParserSession passing in the data supplier, MIME type for WBXML, the MIME type of the document to be parsed and the call-back pointer where parsing events are to be received.

3. The client begins the parsing by calling Start() on the parser session.

4. The parser makes the following call-backs to the client:

    OnStartDocumentL()

    OnStartElementL('A')

    OnStartElementL('B')

    OnContent('Content')

    OnEndElementL('B')

    OnEndElementL('A')

    OnEndDocumentL()

Set-up a parser to parse WBXML with a validator plug-in

The same document as 5.1 is used in this scenario.

1. The client creates a data supplier that contains the data to be parsed.

2. The client constructs a **RMarkupPlugins** object with the UID of a validator.

3. The client creates an **RParserSession** passing in the data supplier, MIME type for WBXML, the MIME type of the document to be parsed, call-back pointer where parsing events are to be received and the array of plug-ins object.

4. The parser session first iterates through the array of plug-ins starting from the end of the list. It creates the **CValidator** ECOM object setting the call back to the client. The **CWbxmlParser** ECOM object is created next and its call-back is set to the **CValidator** object. This sets up the chain of call-back events from the parser through to the validator and then the client. The validator needs access to data from the parser so SetParent needs to be called on all the plug-ins in the array. The validator sets its parent to the parser object.

5. The client begins the parsing by calling Start() on the parser session.

6. The parser makes the following call-backs to the client:

    OnStartDocumentL()

    OnStartElementL('A')

    OnStartElementL('B')

    OnContent('Content')

OnEndElementL('B')

OnEndElementL('A')

OnEndDocumentL()

5   **Generating a WBXML document with a DTD validator**

The document in 5.1 is to be generated in this scenario.

1. The client creates a data supplier with an empty buffer.
2. The client constructs a **RMarkupPlugins** object with the UID of a validator.
10   3. The client creates a **RParserGenerator** passing in the data supplier, MIME type for WBXML, MIME type of the document to be parsed and the array of plug-ins object.
4. The generator session first iterates through the array of plug-ins starting from the end of the list. It creates the **CValidator** ECOM object setting the call back to the client. The **CWbxmlGenerator** ECOM object is created next and its call-back is set to the
15   **CValidator** object. This sets up the chain of call-back events from the generator through to the validator and then the client. The validator needs access to data from the parser so SetParent needs to be called on all the plug-ins in the array. The validator sets its parent to the parser object.
5. The client then calls the following methods:

20   BuildStartDocumentL()

          BuildStartElementL('A')

          BuildStartElementL('B')

          BuildContentL('Content')

          BuildEndElementL('B')

25   BuildEndElementL('A')

Design Considerations
- ROM/RAM Memory Strategy – the string pool is used to minimise duplicate strings. Error condition handling – errors are returned back to plug-ins and the client via the
30   call-back API.
- Localisation issues – documents can use any character set and the character set is returned back to the client in the case of parsing so it knows how to deal with the data. For a generator the client can set the character set of the document.
- Performance considerations – the string pool makes string comparisons efficient.

- Platform Security – in normal usage the parser and generator do not need any capabilities. However, if a plug-in were designed to load a DTD from the Internet it would require PhoneNetwork capabilities.

- Modularity – all components in the framework are ECOM components that can be
5   replaced or added to in the future.

Testing

The data supplier and parser generator set-up components can be tested individually - all the functions are synchronous and therefore no active objects need to be created for testing.

10

The following steps can be carried out to test parsing and generation of WBXML or XML:·

1. Load a pre-created file.
15   2. Parse the file.
    3. Generate a buffer from the output of the parser.
    4. Compare the output of the buffer with the original pre-created file to see if they match.

20   Additional tests are carried out to test error conditions of parsing, such as badly formatted documents and corrupt documents.

Role	Person / People
Contributor(s)	Ian McDowall, Brian Evans, Martin Hardman, David Harper, Lobel Crnogorac, Paul Hadfield, Martin Hardman
Reviewer(s)	Ian McDowall, Brian Evans, Martin Hardman, David Harper, Lobel Crnogorac, Paul Hadfield, Martin Hardman

References

No.	Document Reference	Version	Description
[1]	http: www.w3.org/TR/wd-xml 970807.html	1.0	XML specs
[2]	Binary XML Content Format Specification	1.3	The WBXML specification.

Open Issues

The following issues need to be resolved before this document is completed:

5   1. If a plug-in requires capabilities to connect to the Internet what capabilities does the framework need?

2. The API for **CMarkupCharSetConvertor** and **RDocumentParameters** needs to be decided.

Glossary

10   The following technical terms and abbreviations are used within this document.

Term	Definition
XML	Extensible Markup Language
WBXML	WAP Binary Extensible Markup Language
SAX	Simple API for XML
DOM	Document Object Model
Element	This is a tag enclosed by angle brackets. E.g <Name>, <Address>, <Phone> etc
Attributes	These are the attributes associated with an element. E.g. <Phone Type="Mobile"> The attribute here is "Type".
Values	These are the actual value of an attribute. E.g. <Phone Type="Mobile"> The value here is "Mobile"
Content	This is the actual content for an element. E.g. <Name>Symbian</Name>. Here "Symbian" is the content for the element "Name".
DTD	Document Type Definition
MIME	Multipurpose Internet Mail Extensions
Code Page	Since only 32 elements can be defined In WBXML, code pages are created so that each

Term	Definition
	code page can have 32 elements.
XSLT	Extensible Style-sheet Language Transformations
SOAP	Simple Object Access Protocol
URI	Uniform Resource Identifiers
qualified name	A qualified name specifies a prefix : local name e.g. 'HTML:B'
prefix	From the qualified name example this is 'HTML'
local name	From the qualified name example this is 'B'

Document History

Date	Version	Status	Description
dd-mm-yyyy	1.0	Issued	
20-09-2002	0.4	Draft	Added changes suggested by design meeting.
20-08-2002	0.3	Draft	Updated design.
10-04-2002	0.2	Draft	Added changes suggest by reviewers
1-02-2002	0.1	Draft	First draft based on SGL.PPS001.457 Component Design Document Template Rev 1.0

Document Review Date

Appendix A - <Auto correction examples>

Table A1 shows a situation where the end tags are the wrong way round for A and B. This is very easy to fix since the DTD validator keeps a stack of the tags, it knows what the end tag should be.

5

```
<A>Content
        <B>
        More content
        </A>
</B>
```

Table A1: End tags that are the wrong way round

15   Table A2 shows the situation where the B end tag is missing. Since the end tag does not match a guess can be made that there should be an end tag for B before the end tag of A.

20
```
<A>Content
        <B>
        More content
</A>
```

Table A2: Missing end tag

25

Table A3 shows the situation where there are no end tags for A and B. The DTD validator will detect the problem and send an end tag for B to the client. The auto correct component will query the DTD validator if the C tag is valid for the parent element A. If it is valid a **OnStartELementL()** will be sent to the client, otherwise the auto correct

30   component can check further up the element stack to find where this element is valid. If it is not valid anywhere in the stack then it will be ignored together with any content and end element tag.

```
<A>Content
        <B>
        More content
        <C>
        Some content
        </C>
```

Table A3: Missing end tags

Appendix B - How to write a namespace plug-in

The tables below show the WBXML tokens for the example namespace. Tables 1 to 3 each represent a static string table. Tables 1 shows the elements for code page 0. Tables 2 and 3 are for attribute value pairs respectively. Each attribute index on Table 2 refers to 5   the values of the same index in Table 3. These token values must match up in Tables 2 and 3. If an attribute does not have a value then there must be a blank as shown in Table 3 with token 8. For attribute values, these also appear in Table 3 but have a WBXML token value of 128 or greater.

Element type name	WBXML token
Addr	5
AddType	6
Auth	7
AuthLevel	8

10   *Table 1: ElementTable0, code page 0*

Attribute name/value pair (attribute part)	WBXML token
TYPE	6
TYPE	7
NAME	8
NAME	9 .

Table 2: AttributeValuePairNameTable, code page 0

Attribute name/value pair (value part)	WBXML token
ADDRESS	6
URL	7
	8
BEARER	9
GSM/CSD	128

GSM/SMS	129
GSM/USSD	130

Table 3: AttributeValuePairValueTable, code page 0

The following string table files (.st) are created for each table:

5

Element table for code page 0

stringtable ElementCodePage0

EAddr Addr

EAddType AddType

10   EAuth Auth

EAuthLevel AuthLevel

String table for Table 1

Attributes table for code page 0

15   stringtable AttributesCodePage0

EType Type

EType Type

EName Name

EName Name

20   String table for Table 2

Attribute values table for code page 0

stringtable AttributeValuesCodePage0

EAddress Address

25   EURL URL

EBearer BEARER

EGSM_CSD GSM/CSD

EGSM_SMS GSM/SMS

EGSM_USSD GSM/USSD

30   String Table for Table 3

<Example usage of API >

Below shows an example of how to setting up the parser and generator with DTD
5   checking and auto correction.

```
RMarkupPlugins plugins;
plugins.Append(KMyValidator);
plugins.Append(KMyAutoCorrector};
CDescriptorDataSupplier* dataSupplier = CDescriptorDataSupplier::NewLC();
RParserSession parser;
parser.OpenL(dataSupplier, MarkupMimeType, DocumentMimeType, callback, plugins);
parser.Parse();
// Callback events will be received
parser.Close();


// Now construct a generator using the same plug-ins and data supplier
RGeneratorSession generator;
generator.OpenL(dataSupplier, MarkupMimeType, DocumentMimeType, callback,
plugins);
generator.BuildStartDocumentL();
RAttributeArray attributes;
// Get an RString from the ElementStringTable
RString string=generator.StringPool().String(ElementStringTable::Tag1, ElementStringTable);
// Build one element with content
generator.BuildStartElementL(string, attributes);
generator.BuildContentL(_L8("This is the content"));
generator.BuildEndElementL(string);
generator.BuildEndDocumentL();
generator.Close();
```

Appendix 2

How the String Pool is used to parse both text and binary mark-up language

5   The Mark-up Language framework design relies on the fact that it is possible (using the 'String Pool' techniques described below) to provide the same interface to clients no matter if text or binary mark-up language is used.

Text based mark up languages use strings, i.e. sequences of characters or binary data. In
10   the String Pool technique, static tables of these strings are created at compile time, with one string table per namespace, for all the elements, attributes and attribute values needed to describe a particular type of mark-up document. Each element, attribute and attribute value is assigned an integer number and these integer 'handles' form an index of the strings.   A string in an XML document can be rapidly compared to all strings in the
15   string table by the efficient process of comparing the integer representation of the string with all of the integer handles in the static string table. The main benefit of using a string pool for parsing is therefore that it makes it very easy and efficient for the client to check for what is being parsed, since handles to strings are used instead of actual strings. This means only integers are compared rather than many characters, as would be the normal
20   case if string pools were not used. Also, comparisons can be carried out in a simple switch statement in the code, making the code efficient, and easier to read and maintain. Hence, the string pool is used to make string comparisons efficient at the expense of creation of the strings.

25   For binary mark-up language (e.g. WBXML) the situation is more complex since there are no strings in WBXML. In WBXML, everything is tokenised (i.e. given a token number). We get around the absence of strings as follows: a table of mappings of each of the WBXML tokens to the index of the string in the string table is created (see **Figure 8**). Each mapping is given a unique integer value – a handle. Since it is required to map
30   from tokens to strings and vice versa, two lists of integer value handles are created: one indexed on tokens and the other indexed on the index of the position in the string table. This is so that it is quick to map from one type to the other. All this is encapsulated in the namespace plug-in and therefore is insulated from the client, parser and generator. The client can therefore parse a binary or text document without having to know about

the specific format – it simply uses the integer handle (RString), which will work correctly for both text and binary mark-up languages.

CLAIMS

1.   A portable computing device programmed with a mark-up language parser or generator that can access data from a source using a generic data supplier API.

2.   The device of Claim 1 in which the source is a buffer in memory.

3.   The device of Claim 1 in which the source is a file.

4.   The device of Claim 1 in which the source is a socket outputting streaming data.

5.   The device of any preceding claim which uses a data source different from that data sources which the device was capable of using when first operated by an end-user.

ABSTRACT

MARK-UP LANGUAGE FRAMEWORK WITH GENERIC HANDLING OF DATA SOURCES

5

A portable computing device programmed with a mark-up language parser or generator that can access data from a source using a generic data supplier API. Hence, the parser or generator is insulated from having to talk directly to a data source; instead, it does so via a generic data supplier API, acting as an intermediary layer. This de-couples the

10 parser or generator from the data source and hence means that the parser or generator no longer have to be hard coded for a specific data supplier. This in turn leads to a simplification of the parser and generator design. The present invention allows parsing and generation to be carried out with any data source. For example, a buffer in memory could be used, as could a file, as could streaming from a socket (hence enabling the

15 ability to parse in real-time from data streamed over the internet).
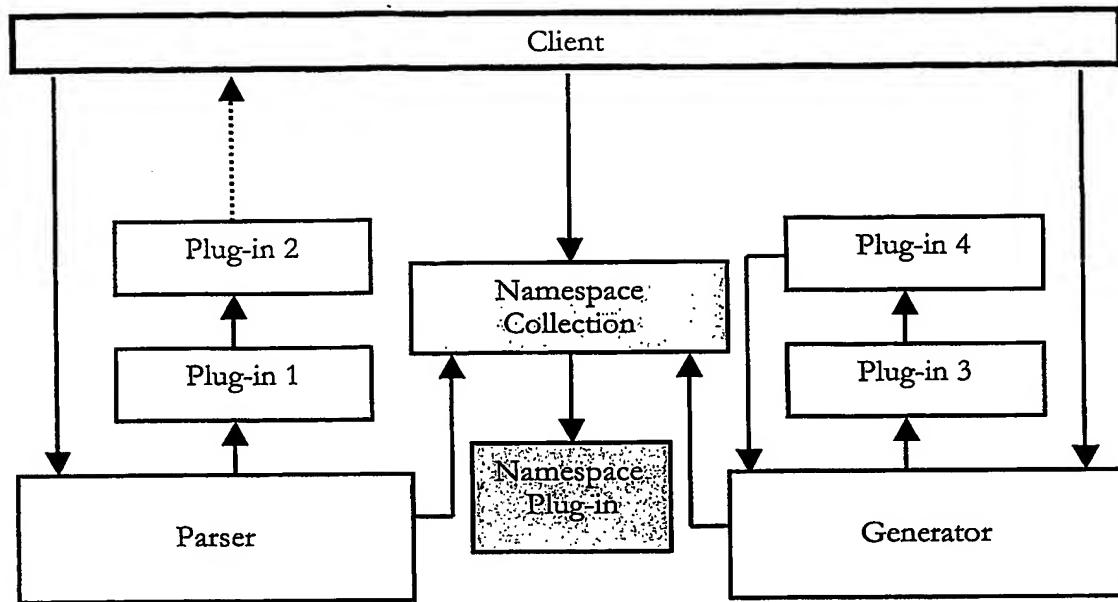
Fig 1: Block diagram of mark-up framework with four plug-ins.
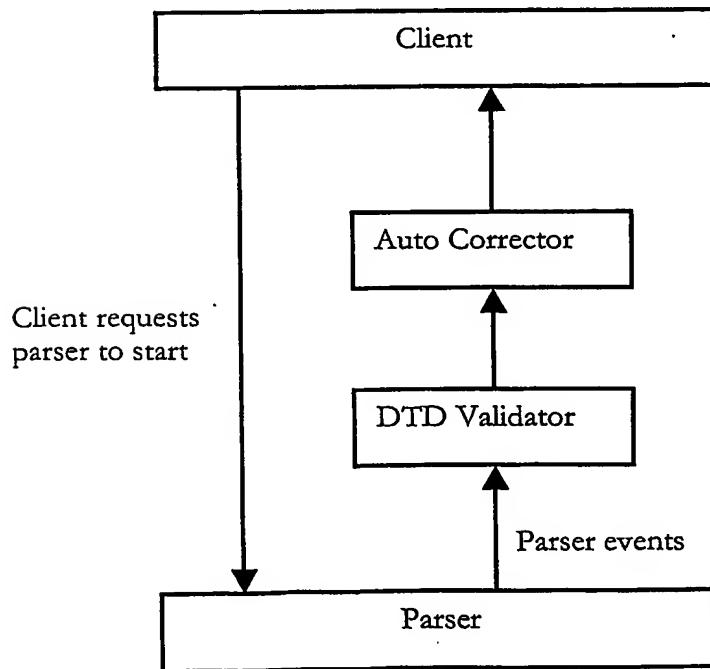
Fig 2: Block diagram of a client parsing using a DTD validator and auto corrector.
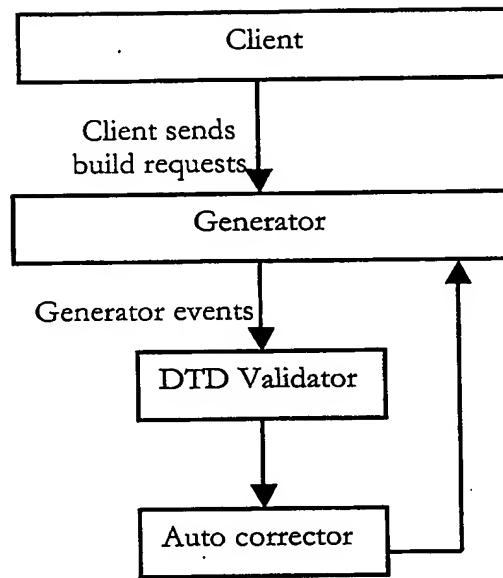
Fig 3: Block diagram of a client using a generator with a DTD validator and auto corrector.
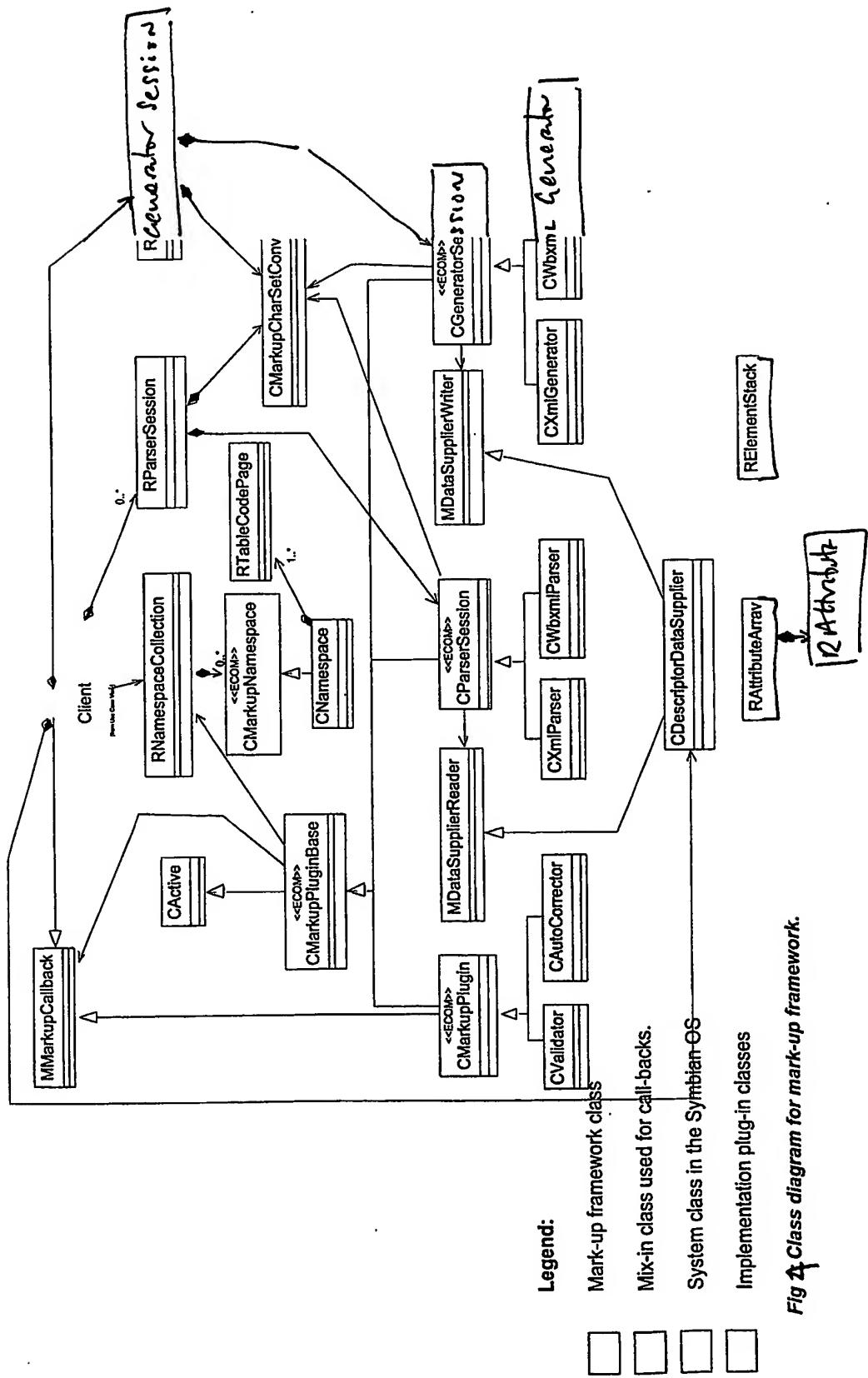
Fig 4 Class diagram for mark-up framework.

Legend:

Mark-up framework class

Mix-in class used for call-backs.

System class in the Symbian OS

Implementation plug-in classes
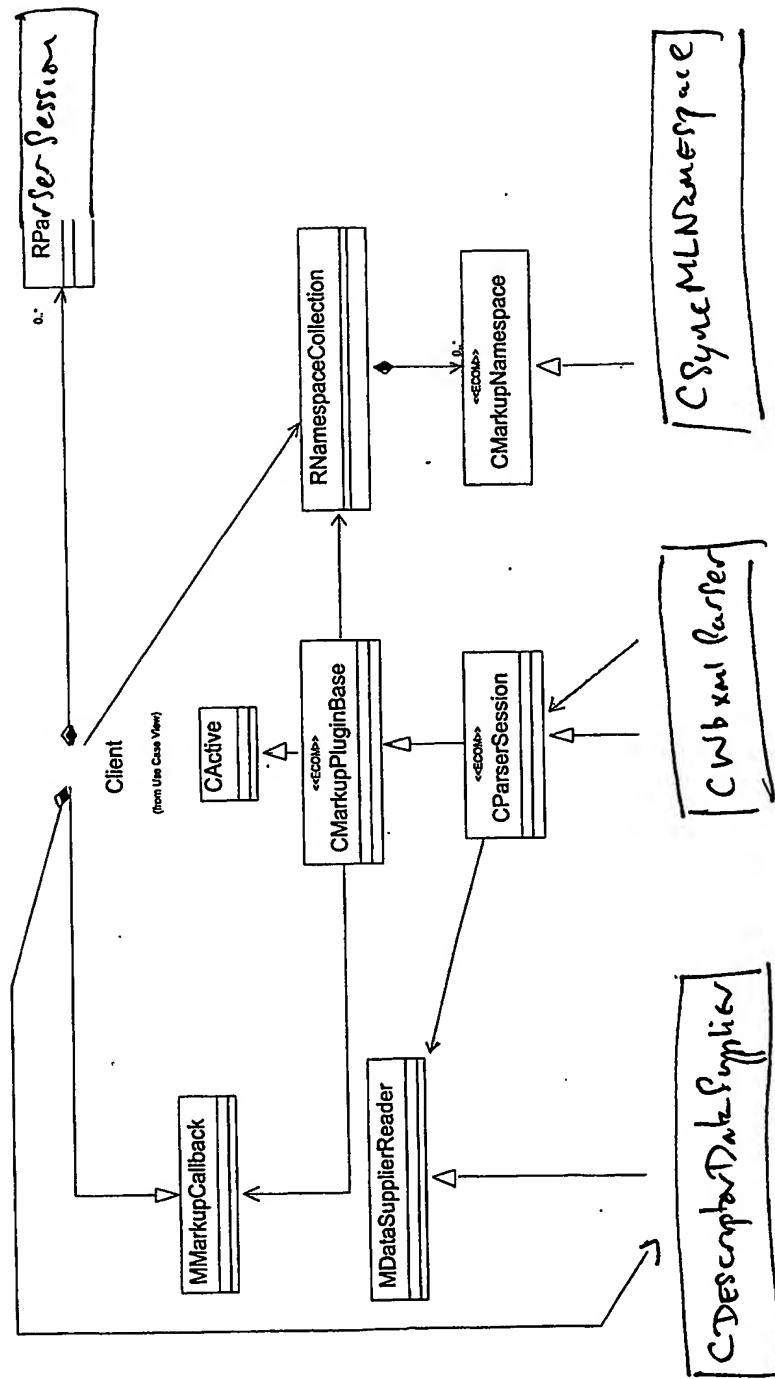
Fig 2: Class diagram of WBXML parser used to parse SyncML using the SyncML namespace
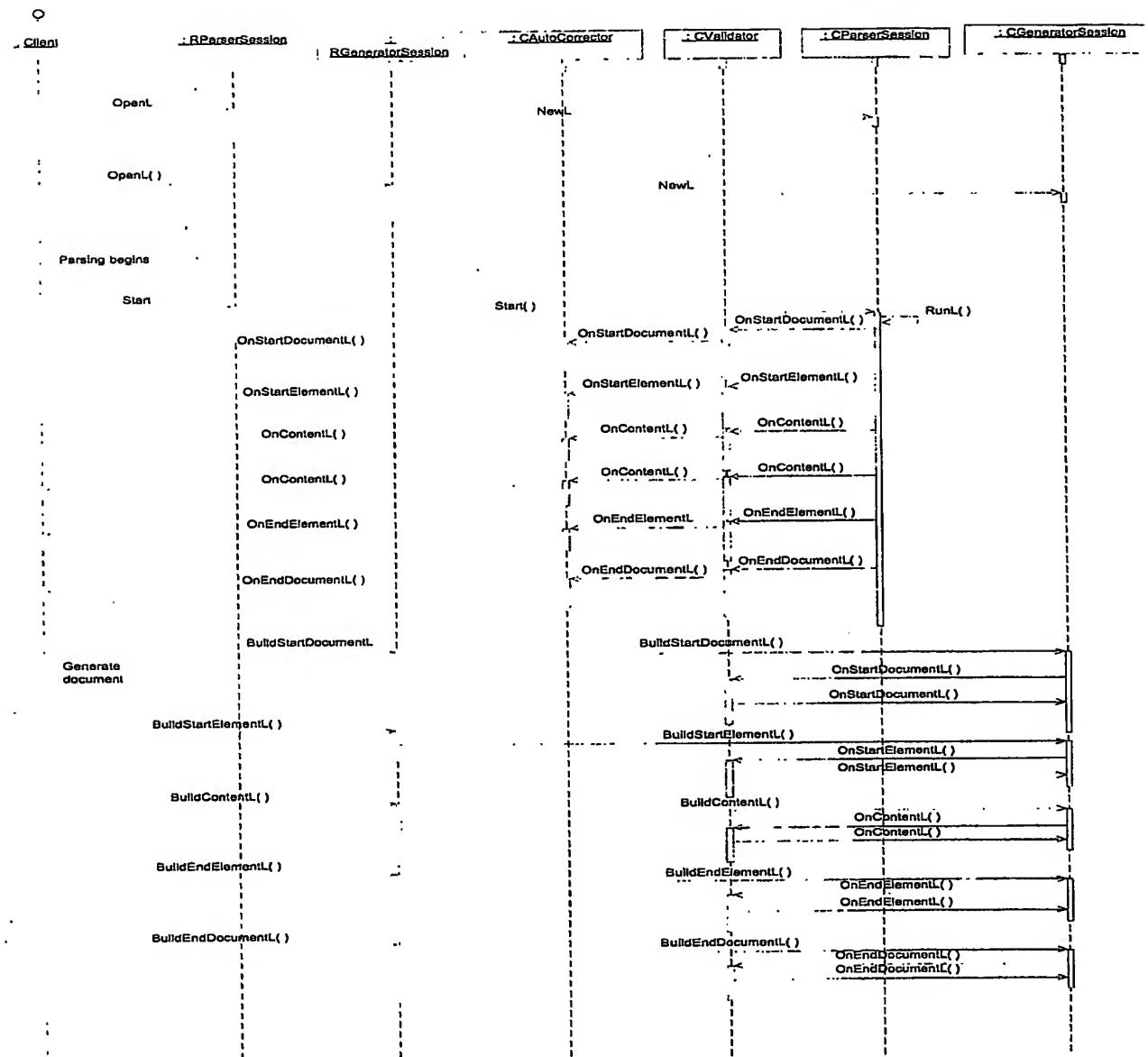
Fig 6: Sequence diagram for parser and generator session

Fig 7: Sequence diagram showing DTD validation and auto correction
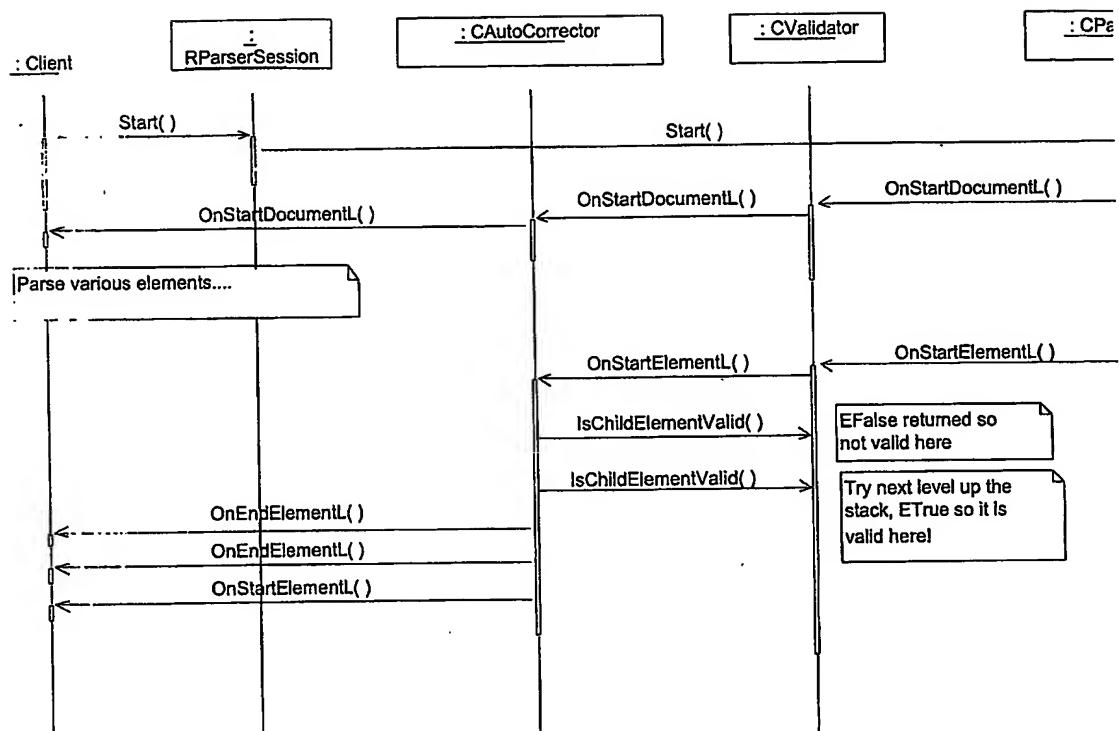
Fig 8: WBXML token of elements mapping to string table of elements in the namespace plug-in.

Element type name		WBXML token
Addr	◄	5
AddType	◄	6
Auth	◄	7
AuthLevel	◄	8